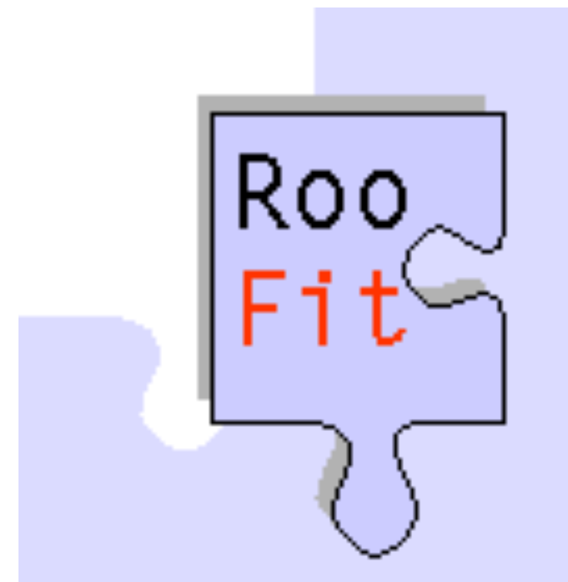
netherlands eScience center



Faster RooFitting

Automated Parallel Computation of Collaborative Statistical Models

Patrick Bos

ACAT 2019, Saas Fee, 14 March

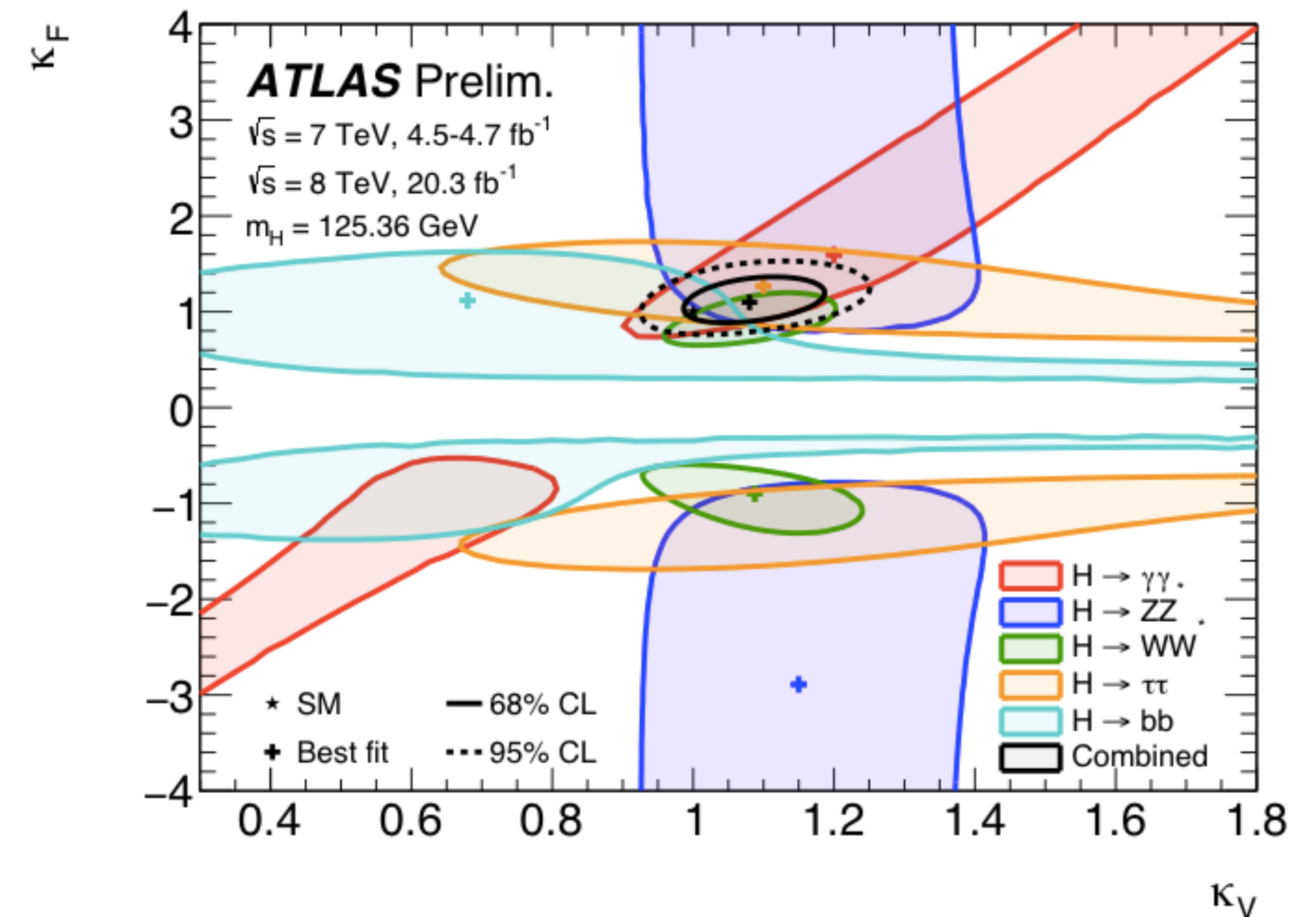
Physics: Wouter **Verkerke** (PI), Vince **Croft**, Carsten **Burgard**

eScience: Patrick **Bos** (yours truly), Inti **Pelupessy**, Jisk **Attema**

RooFit: Collaborative Statistical Modeling

RooFit does statistical inference:

- Build model PDF, combine with data → likelihood L(**data** | a, b...)
- Fit parameters to data; minimization of L w.r.t. parameters (a, b...)



CPU time dominated by (repeated) calculation of L(x)

RooFit: build models **together**

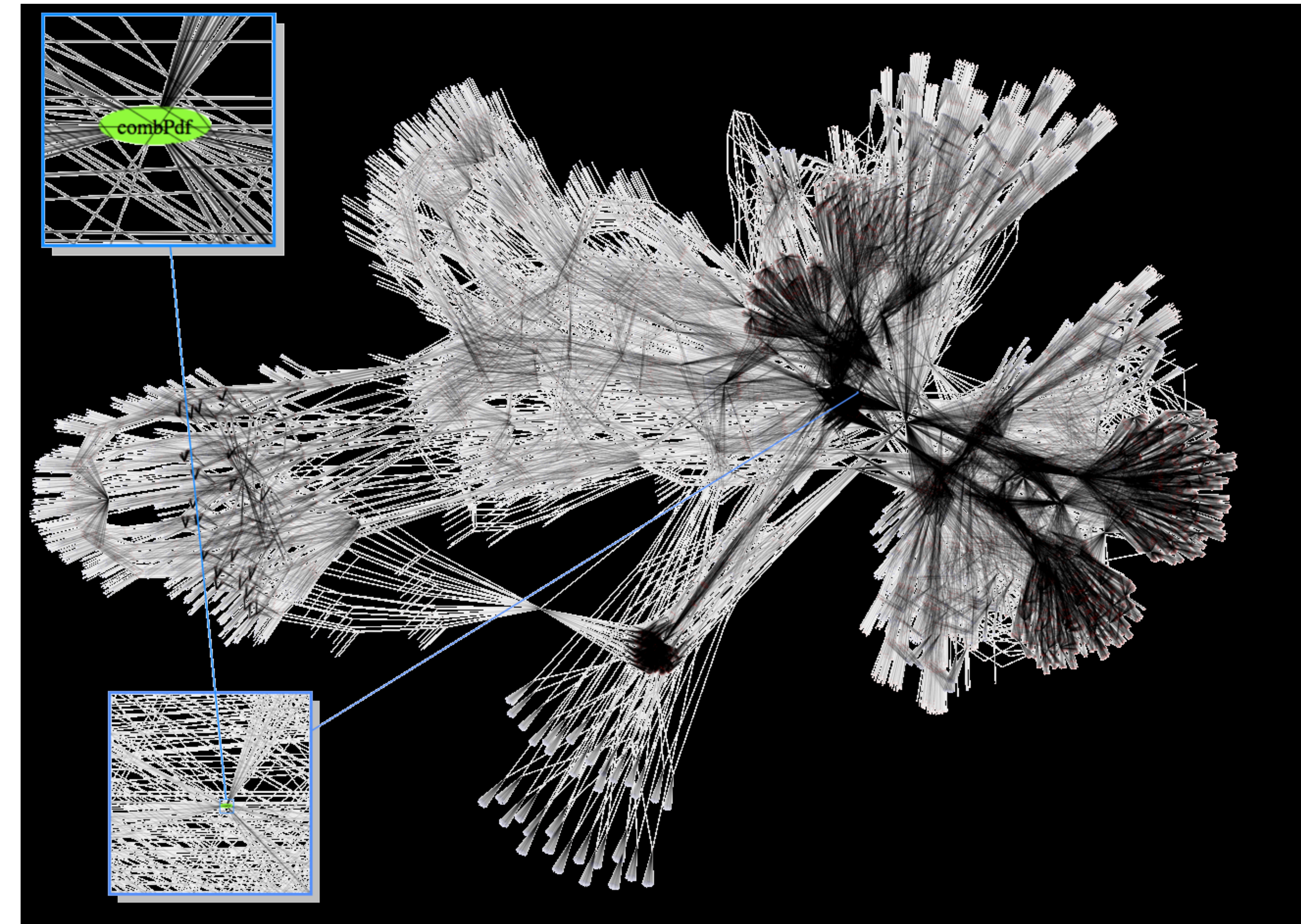
- Teams 10-100 physicists
- Collaborations ~3000

  → ~100 teams

- 1 goal
- (Pretty impressive to an outsider)

Why make RooFit faster

- More **efficient collaboration**
  - Faster iteration/debugging
  - Faster feedback between teams
- Make complex likelihoods fast enough to retain interactive analysis process, e.g.
  - Higgs fit to all channels, ~200 datasets, O(1000) parameter, now O(few) hours
  - EFT framework: again 10-100x more expensive
- Focus on long calculations
  - ~hour fits/minimizations → minutes
  - wall-time speedup goal: x30



Higgs @ ATLAS
20k+ nodes, 125k hours

Expression tree of C++ objects for mathematical components (variables, operators, functions, integrals, datasets, etc.)
Couple with data, event “observables”

Goals and Design:
Make fitting in RooFit faster using (automated) parallel calculation

Existing optimizations:

- pre-calculation/memoization
- likelihood parallelization (but limited)

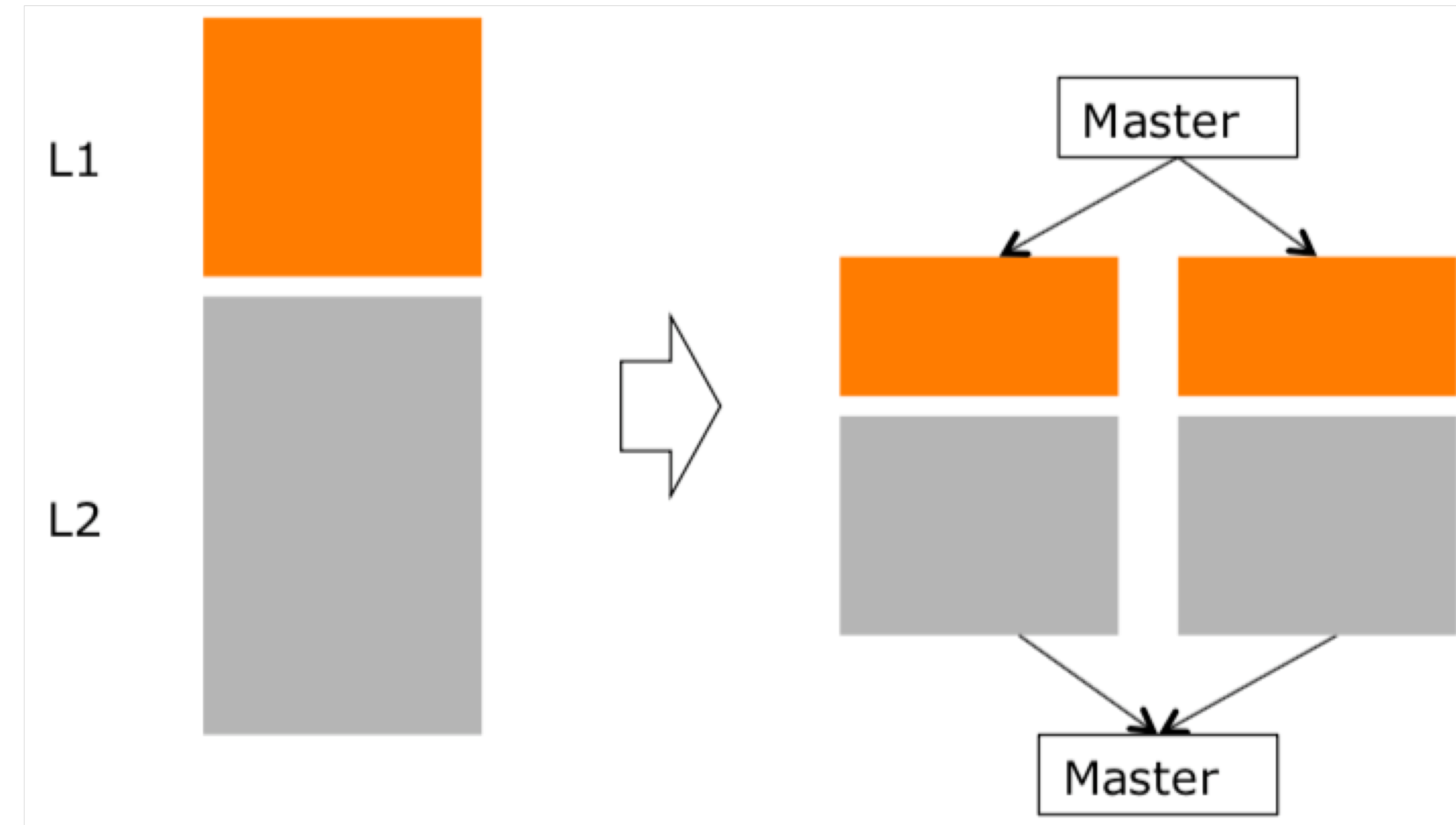
Focus of this project: further **parallelization**

Parallel project: **vectorization** (poster Stephan Hageböck)

Existing parallelization strategy

Existing RooFit likelihood parallelization

Existing parallelization strategy: calculate 1/Nth of each likelihood component on each core.



works well for unbinned ML fits with many events

not so well with:

- expensive num. integrals (calculated on each node)
- many small likelihood components (overhead dominates)
- certain binned fits (e.g. Higgs)

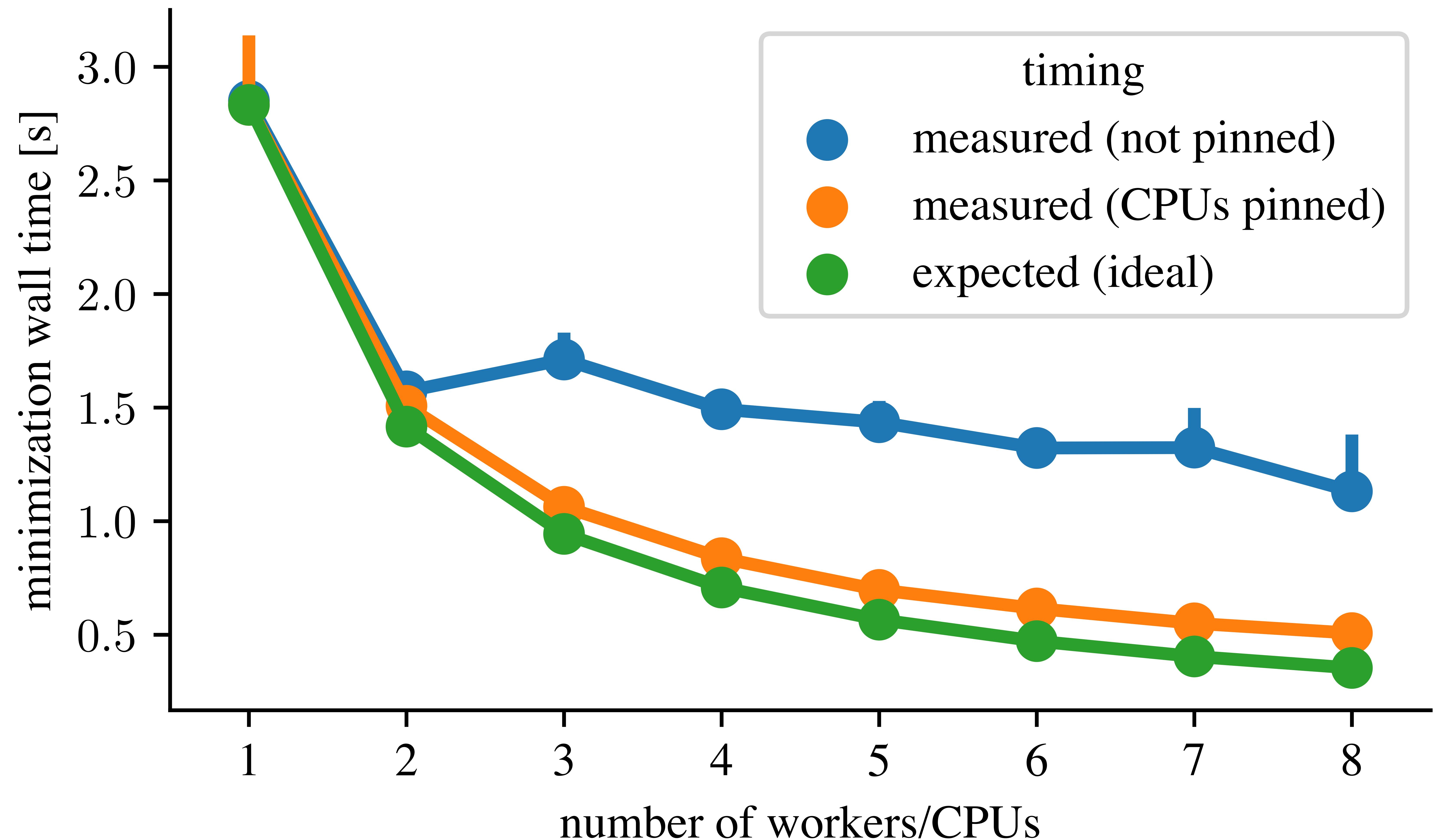
Existing RooFit likelihood parallelization: unbinned, MPFE

existing parallelization performance (1/Nth on each core) for simple likelihood:
unbinned n-D Gaussian likelihood with many events

Improved scaling

Before: max ~2x speedup

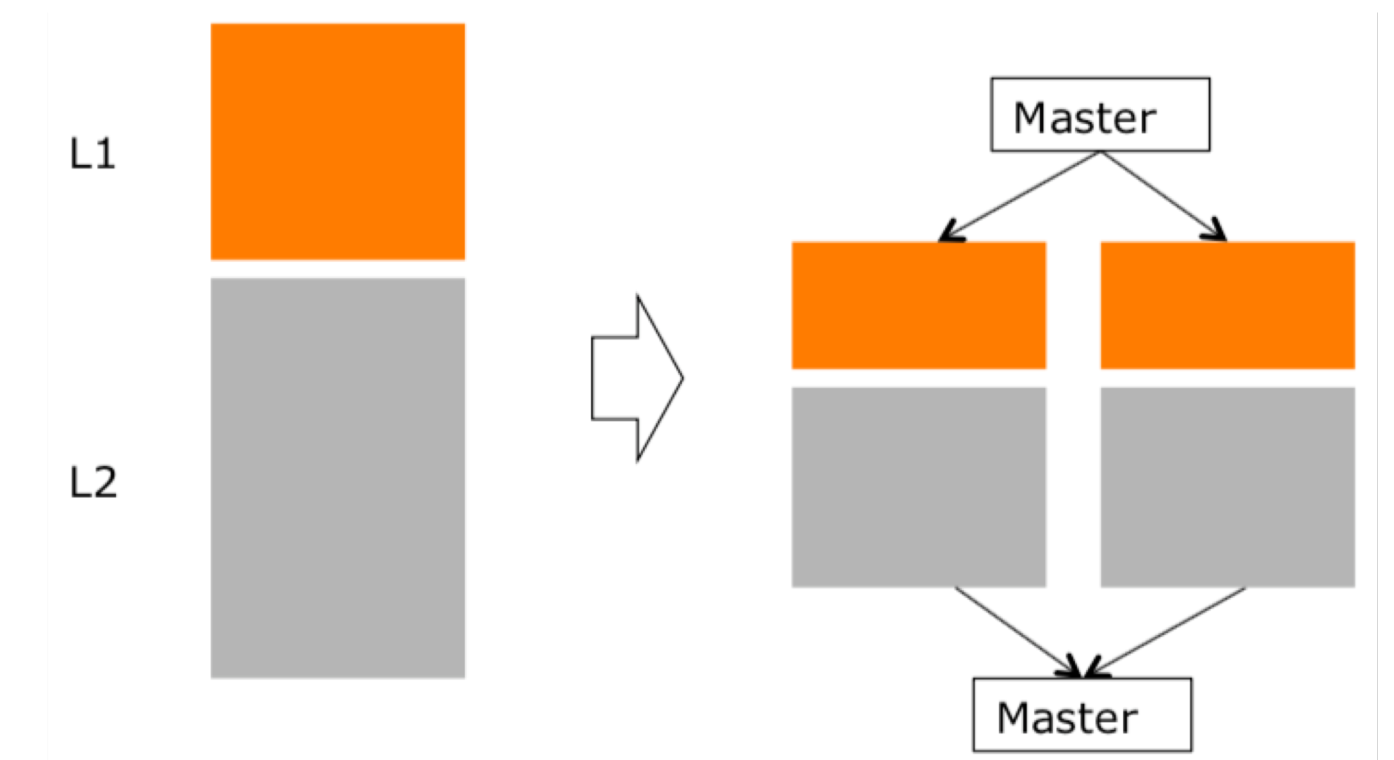
Now (with CPU pinning fixed): max ~20x speedup (more for larger fits)

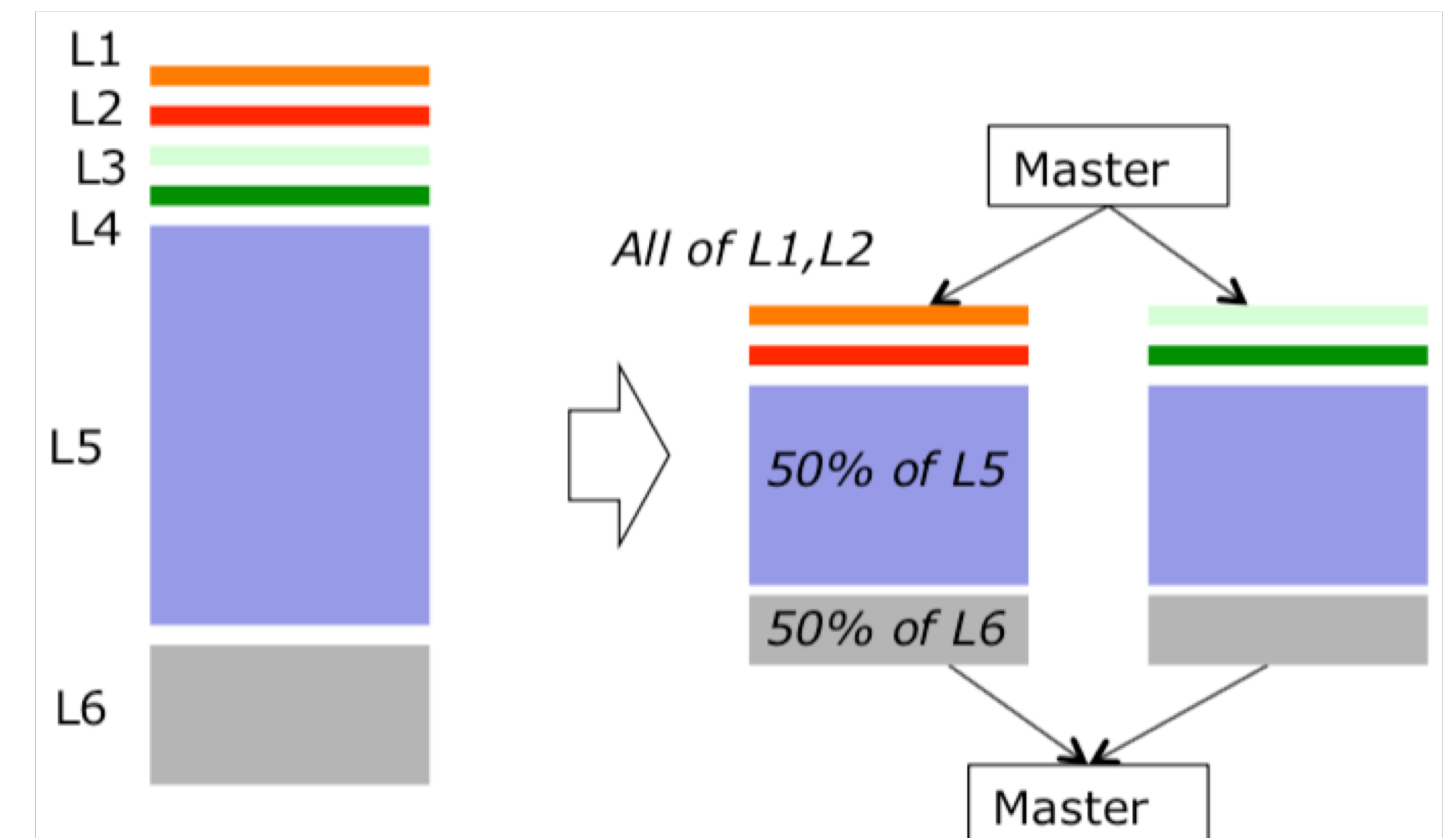


Improving RooFit parallelization for harder cases

Focus on heterogeneous likelihoods (most common in LHC fits)

unbinned likelihood with many events



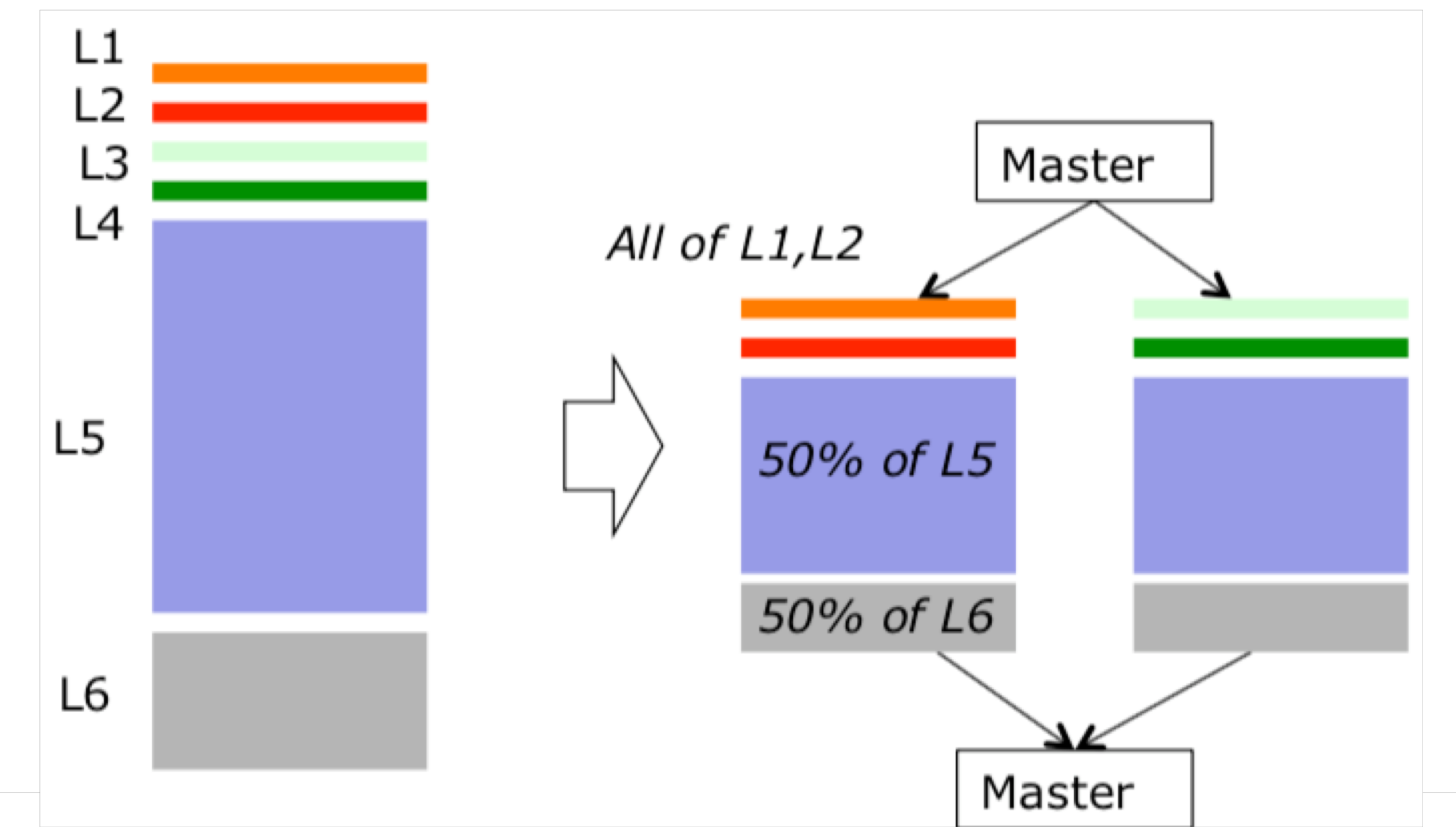


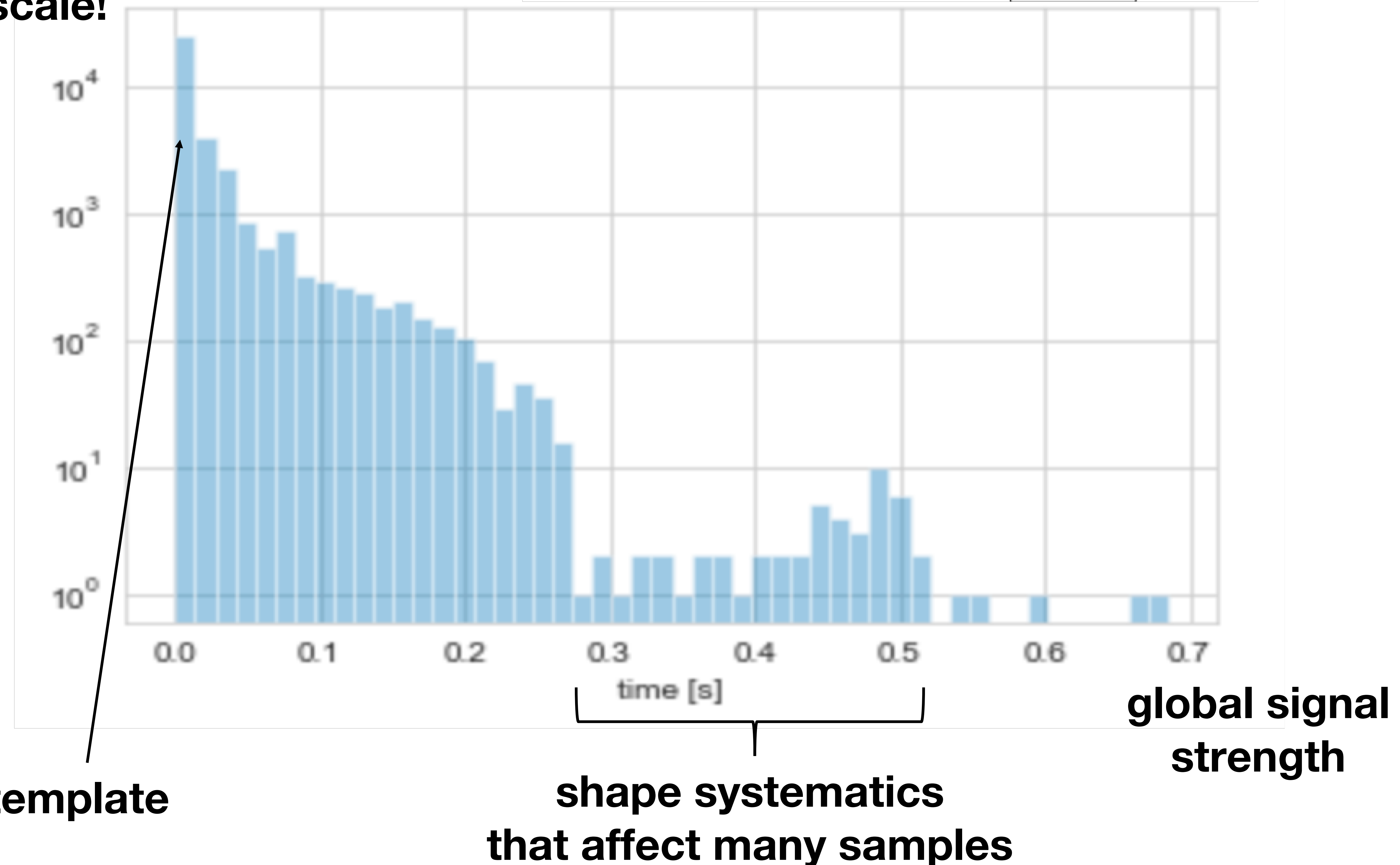
heterogen. likelihood: O(100) unequal dataset components

Heterogeneous models: what are the scaling problems in **existing** strategy?

- Many components too small to efficiently distribute over many cores
  - Relatively large overhead
- Calculation times vary significantly
  - Some components (e.g. binned) will not split by event → all on 1 worker
  - Poor load balancing

heterogen. likelihood: O(100) unequal dataset components



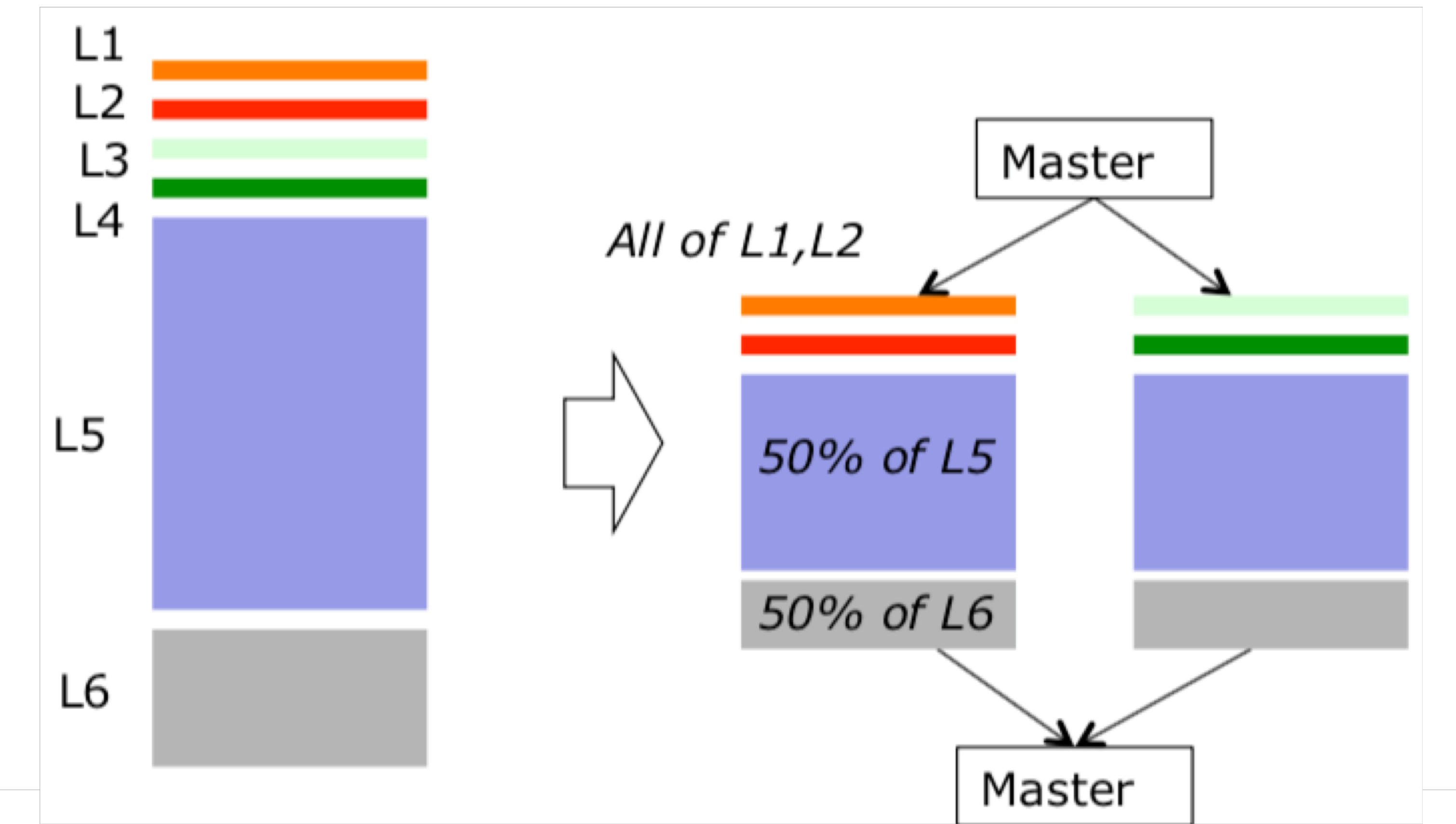


Heterogeneous models:

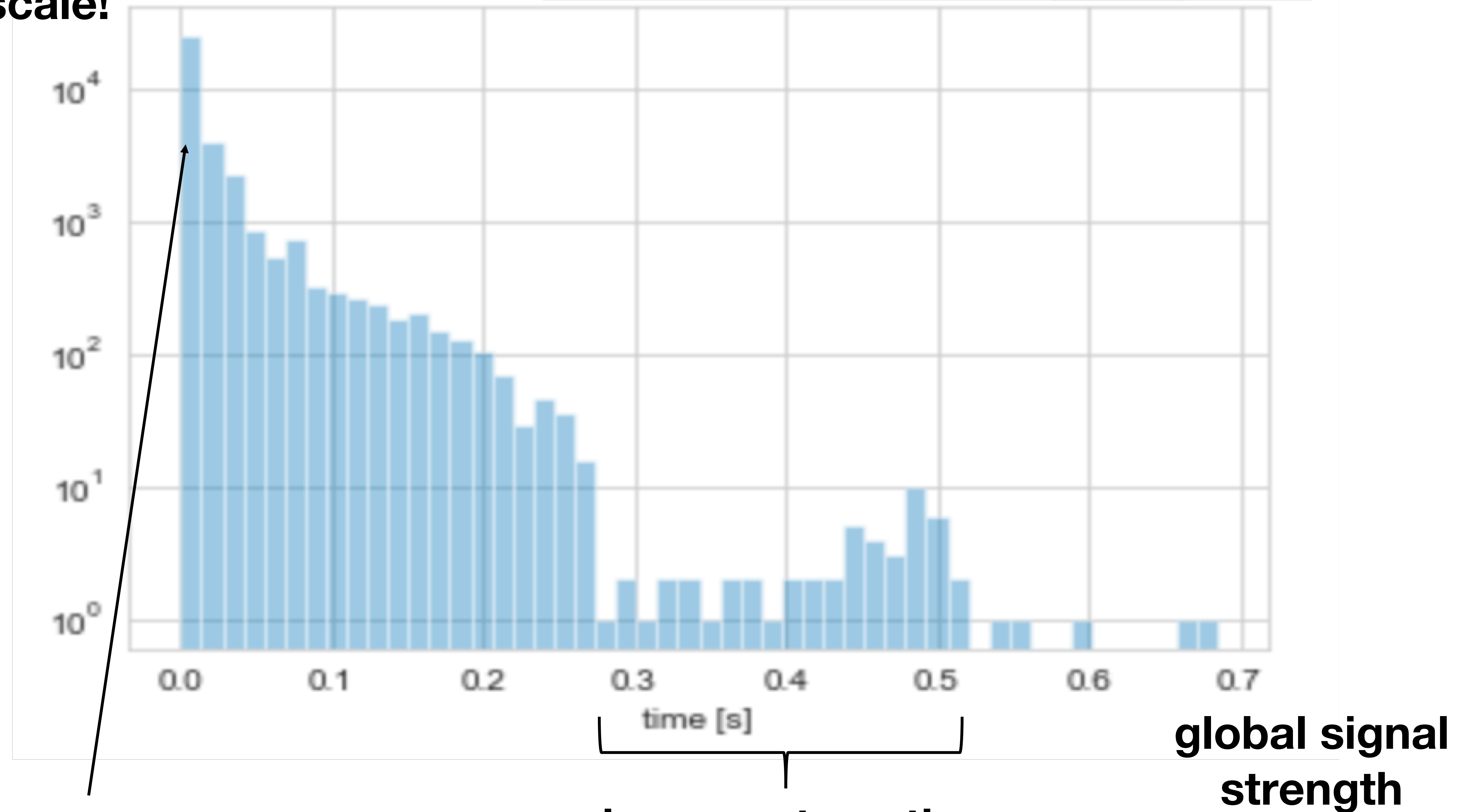
Solutions

- Increase chunk sizes
- Dynamic load balancing

heterogen. likelihood:
O(100) unequal
dataset components



log scale!

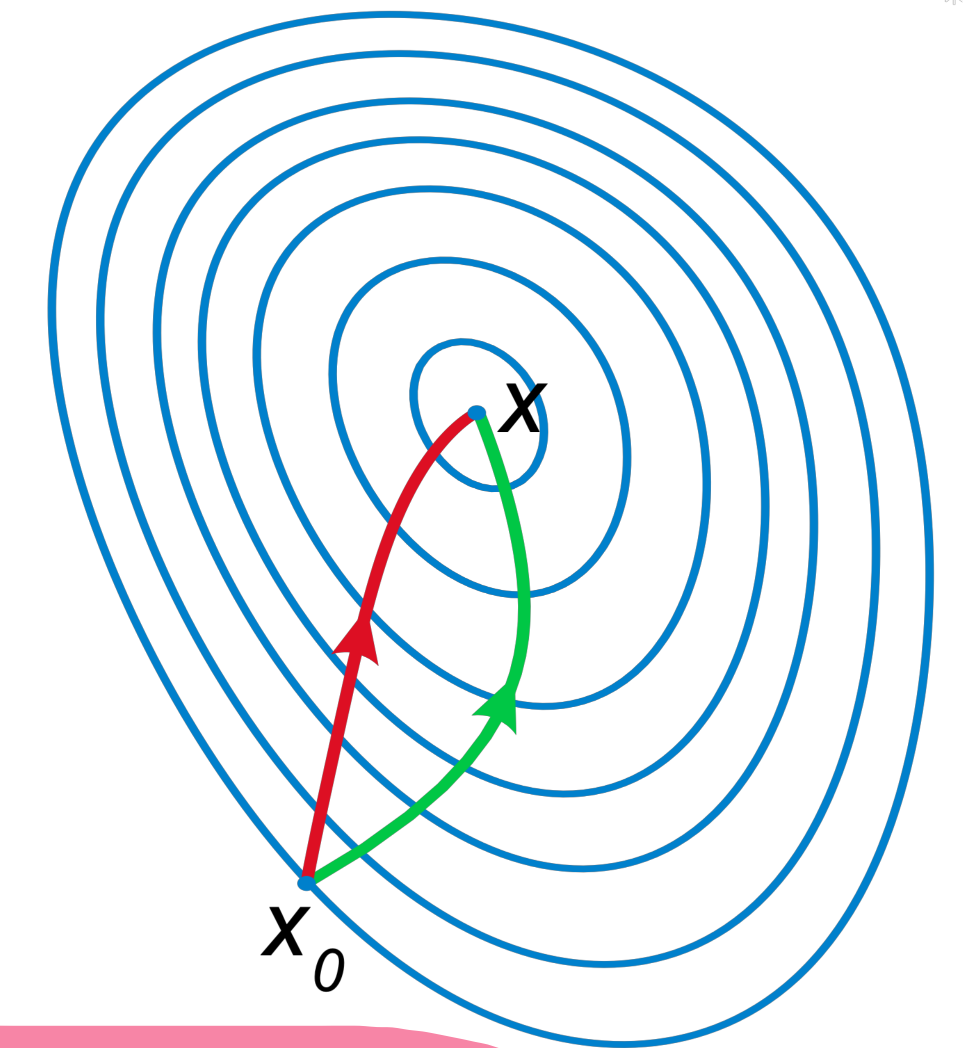


MC stats in 1 bin of template

shape systematics that affect many samples

global signal strength

New parallelization strategies



Most CPU time spent in derivative calculations inside minimizer `MINUIT`

- 1 Iteration ~ Gradient + line-search:
  - gradient for N parameters p: $\frac{df}{dp} \approx \frac{f(p-dp)-f(p)}{dp}$ — 2N f calls → parallelize $\frac{df}{dp}$
  - line-search: descend along gradient direction — 2-3 f calls → parallelize f
- **Focus on partial derivatives** as 'calculation chunk' instead of component likelihood
- Required changes in MINUIT
  - made sure outputs stay exactly same

Dynamic load balancing: MultiProcess task-stealing framework

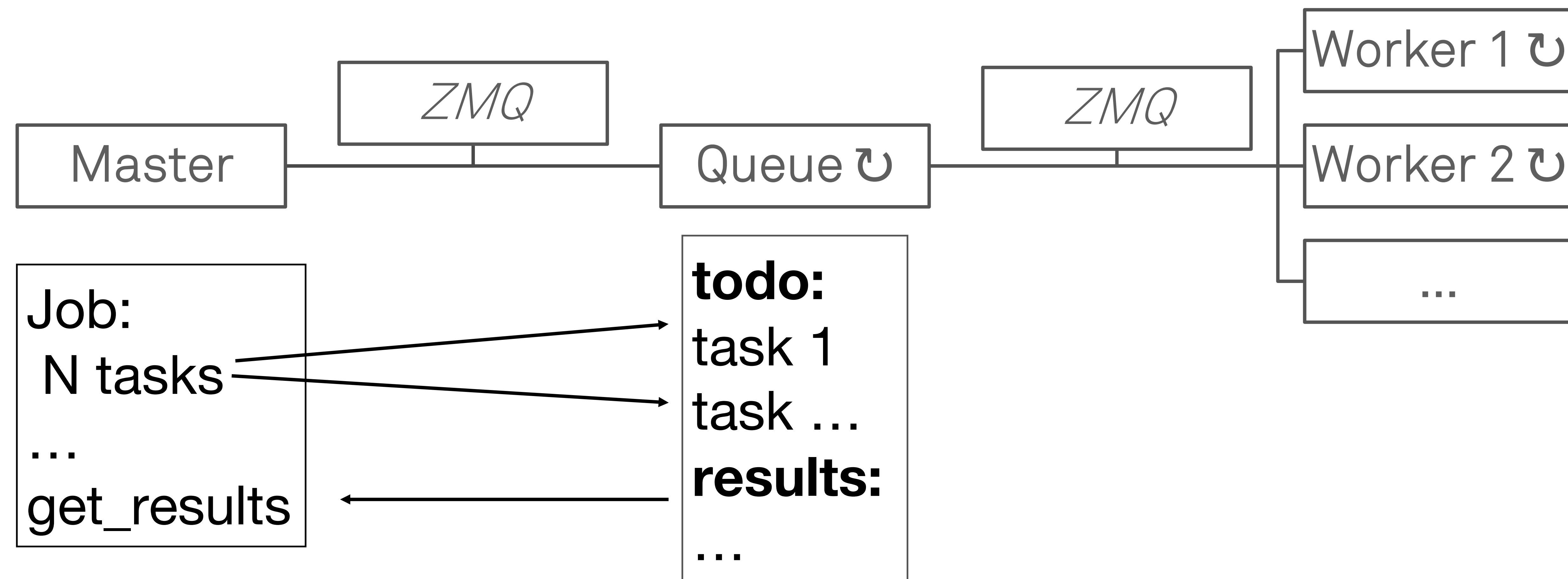
Task-stealing, worker pool, executes **Job** tasks

Job = likelihood component, $\frac{df}{dp}$, …

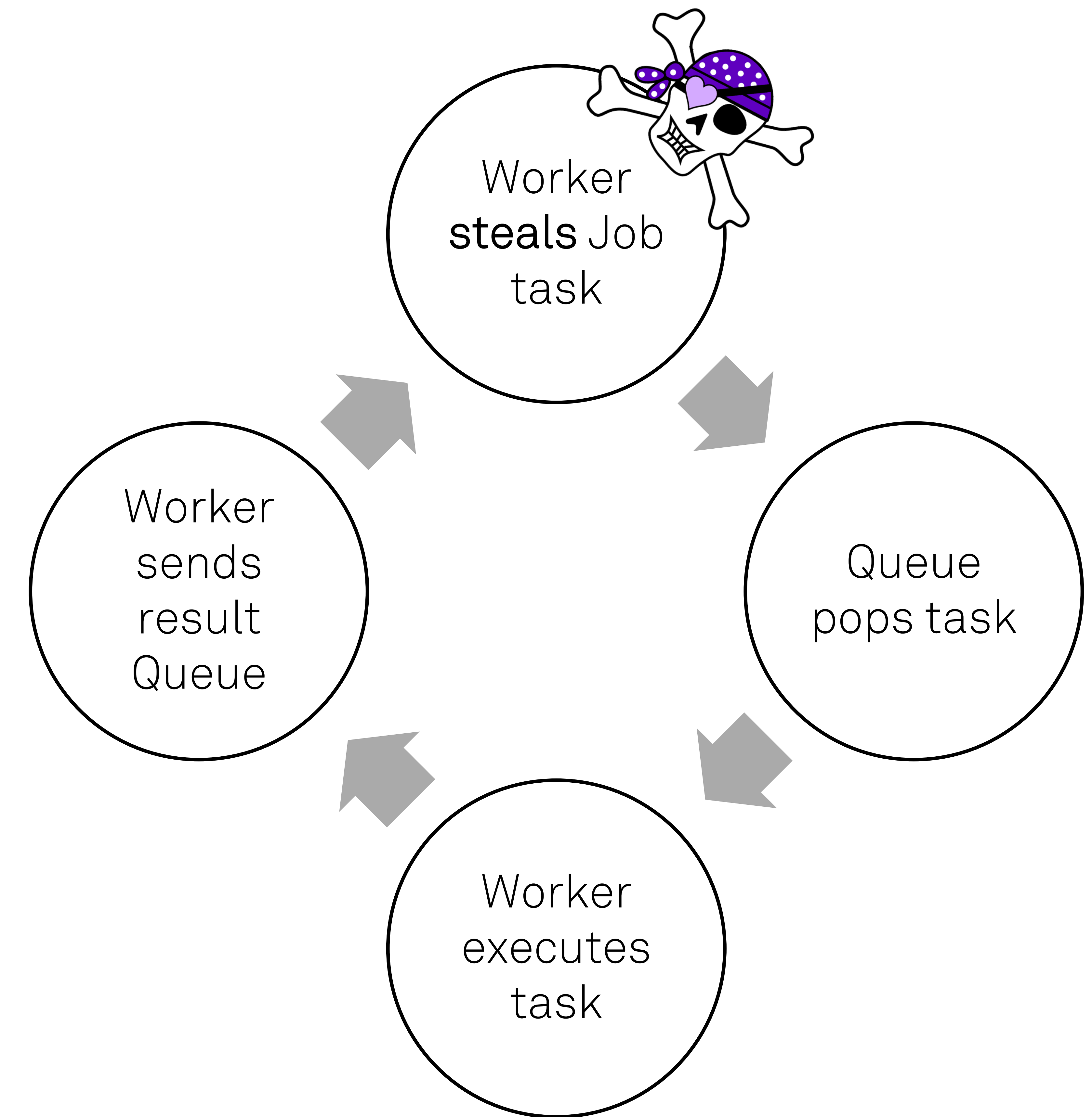
No threads, process-based:

old: custom `BidirMMapPipe` handles fork, mmap, pipes

new: ØMQ for communication between forks



Worker loop:



...until Job done
then Queue sends results
back to Master on request

Parallel gradient performance benchmarks

`MIGRAD` fit / minimization;

Two realistic models:

1. **'fast':** ATLAS H → WW fit

   ~20 seconds -- not main target audience, but used for fast benchmarking

   13795 likelihood components, **265 parameters**

2. **'big':** ATLAS Higgs combination Moriond 2019

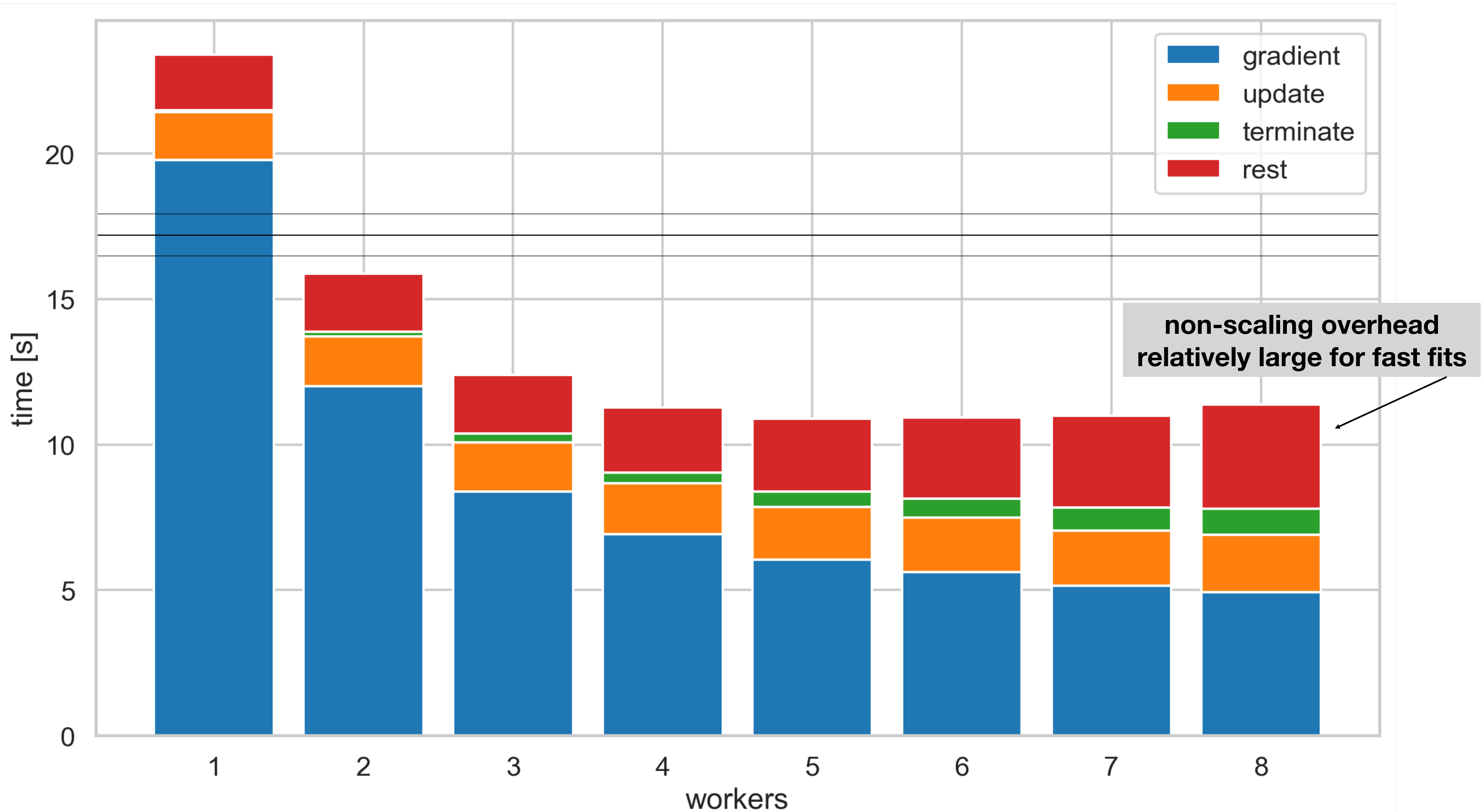
   ~10 minutes to few hours (dep. on distance starting point to minimum)

   126883 likelihood components, **1487 parameters**

- How does Minuit work - and where does it spend it's time
  - Setup/initialize
  - Iterate
    - Gradient calculation (parallized in RooFit)
    - Line search
  - Finalize
- We time separately per iteration
- Relative impact of setup & finalize very dep. on fit, but generally decreasing with number of fit steps

Fast model

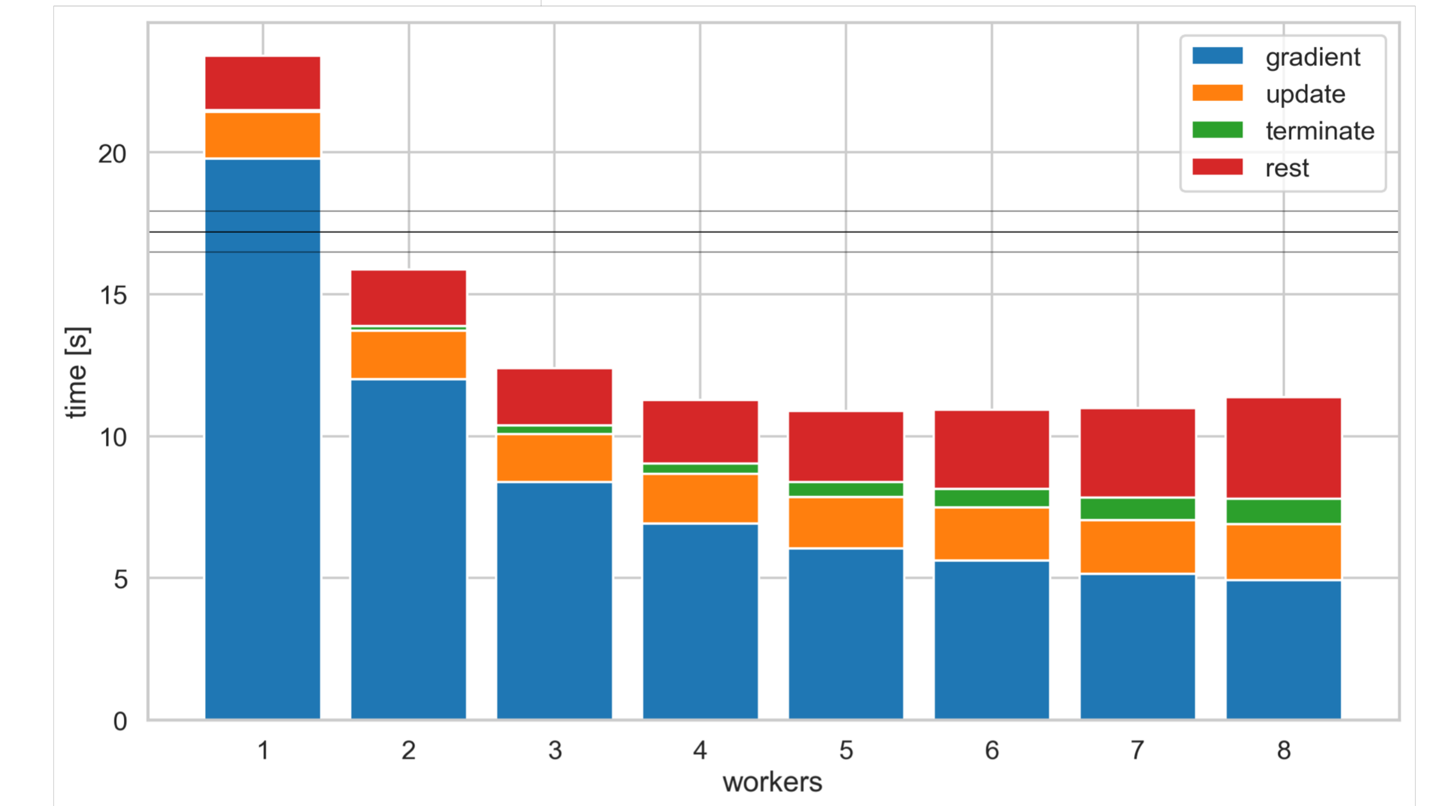
Scaling results on fast model



Scaling results on fast model

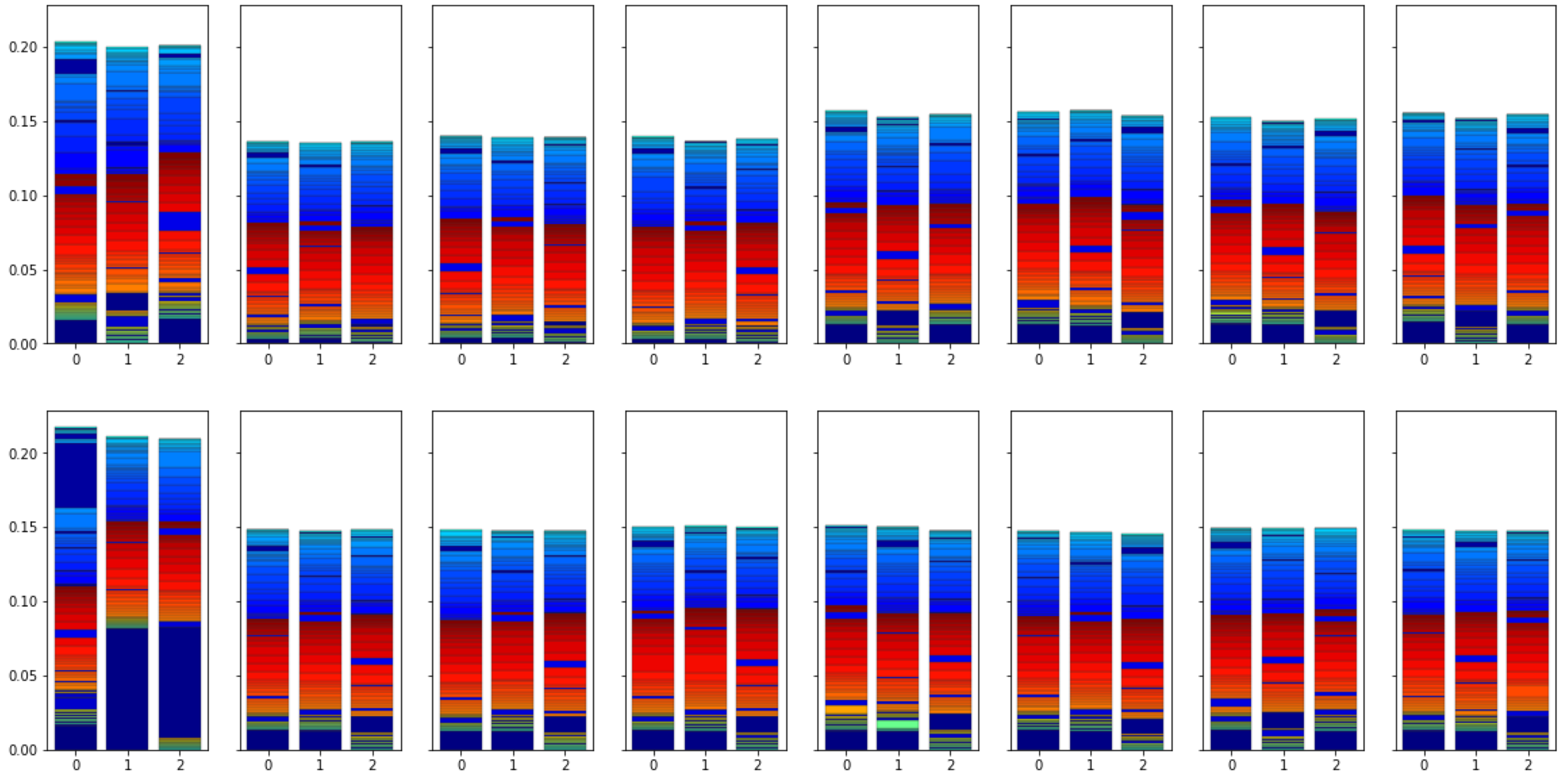
speedup

workers



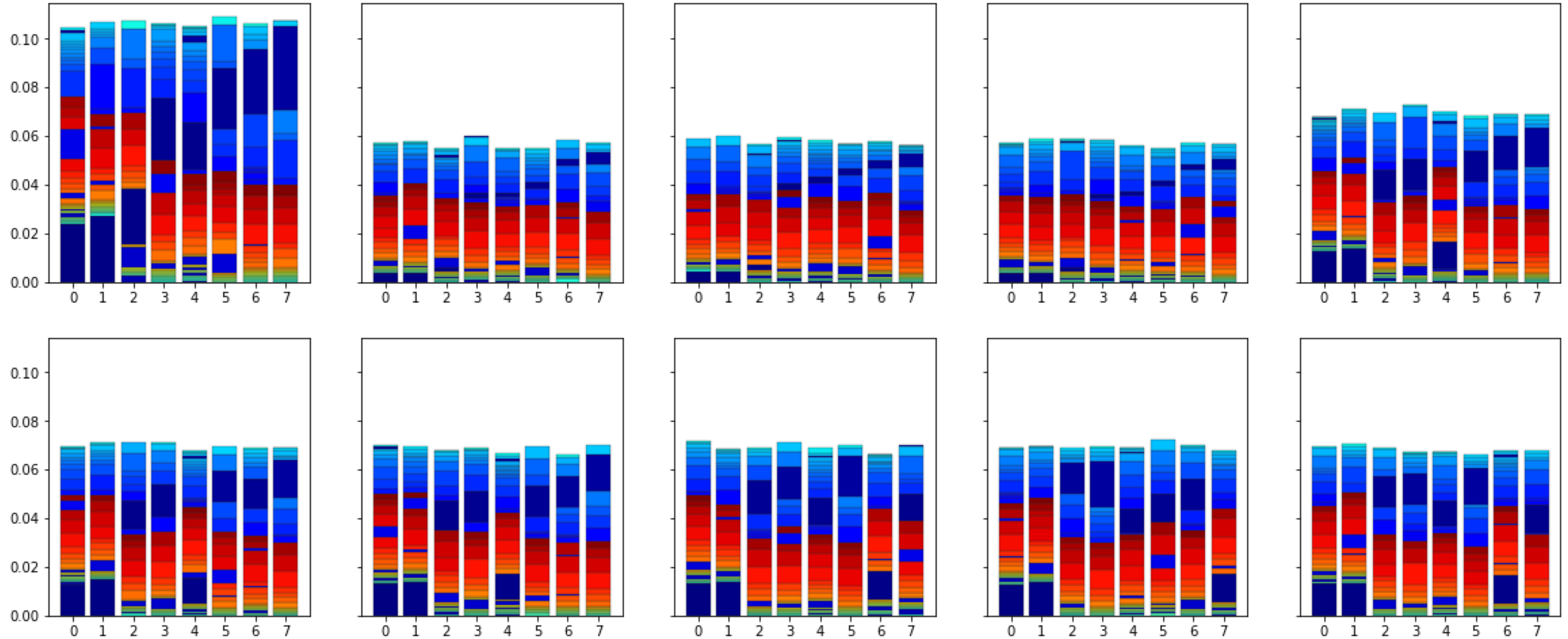
Load balancing results on fast model

3 workers --- panel: one gradient calculation (variable metric step) --- bar = time per partial derivative (task)



Load balancing results on fast model

8 workers --- panel: one gradient calculation (variable metric step) --- bar = time per partial derivative (task)



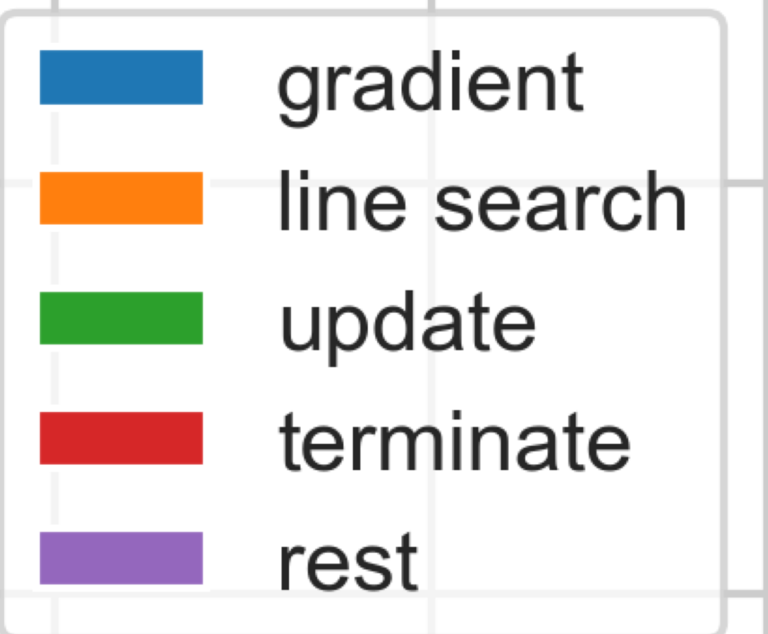
Big model

Scaling results on big model

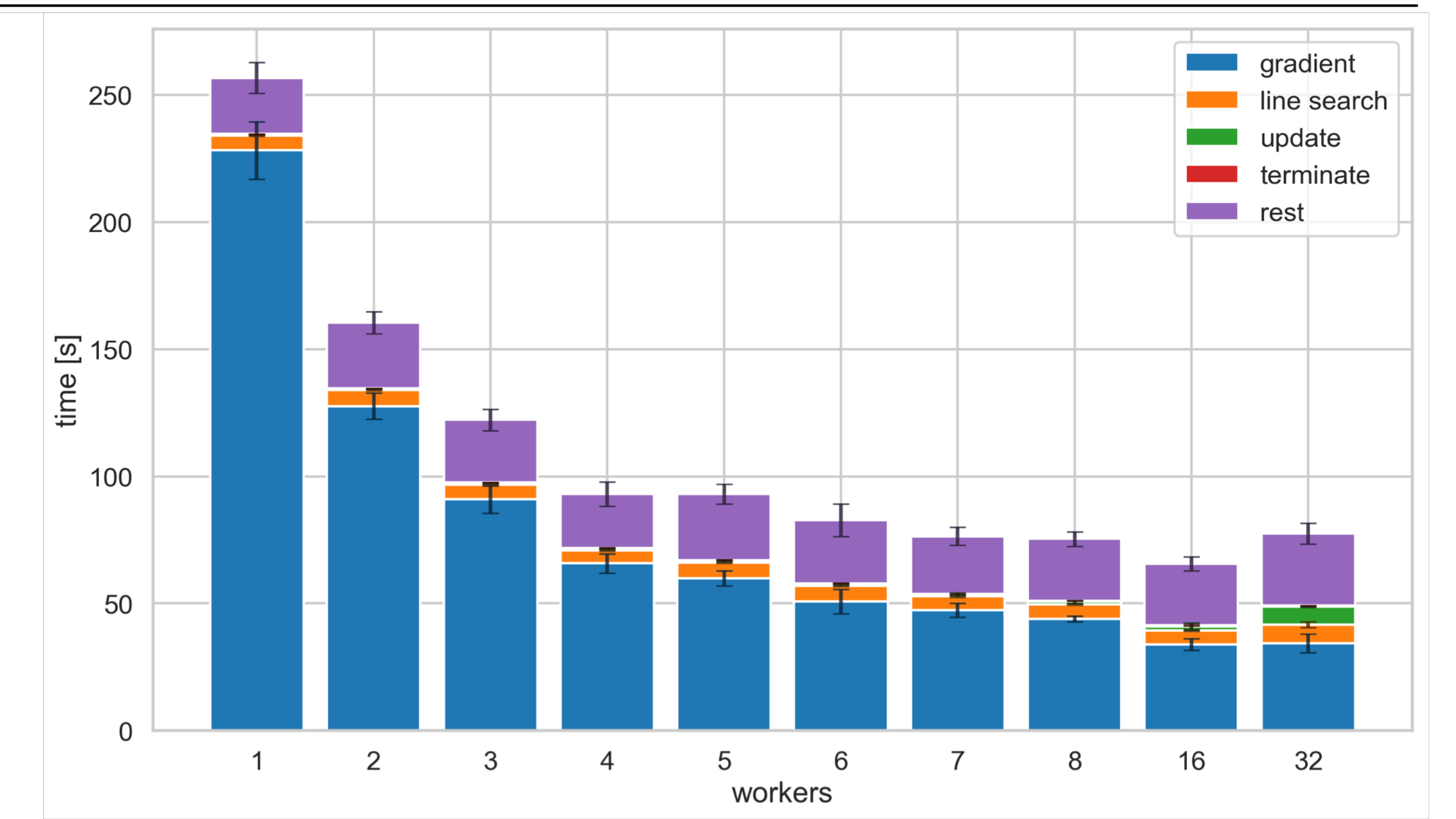
fast run of big model (start very close to minimum)

"rest" overhead indep. of number of iterations (measured this)

this plot: "worst case scenario"



Scaling results on big model



speedup

0 1 2 3 4 5 6 7

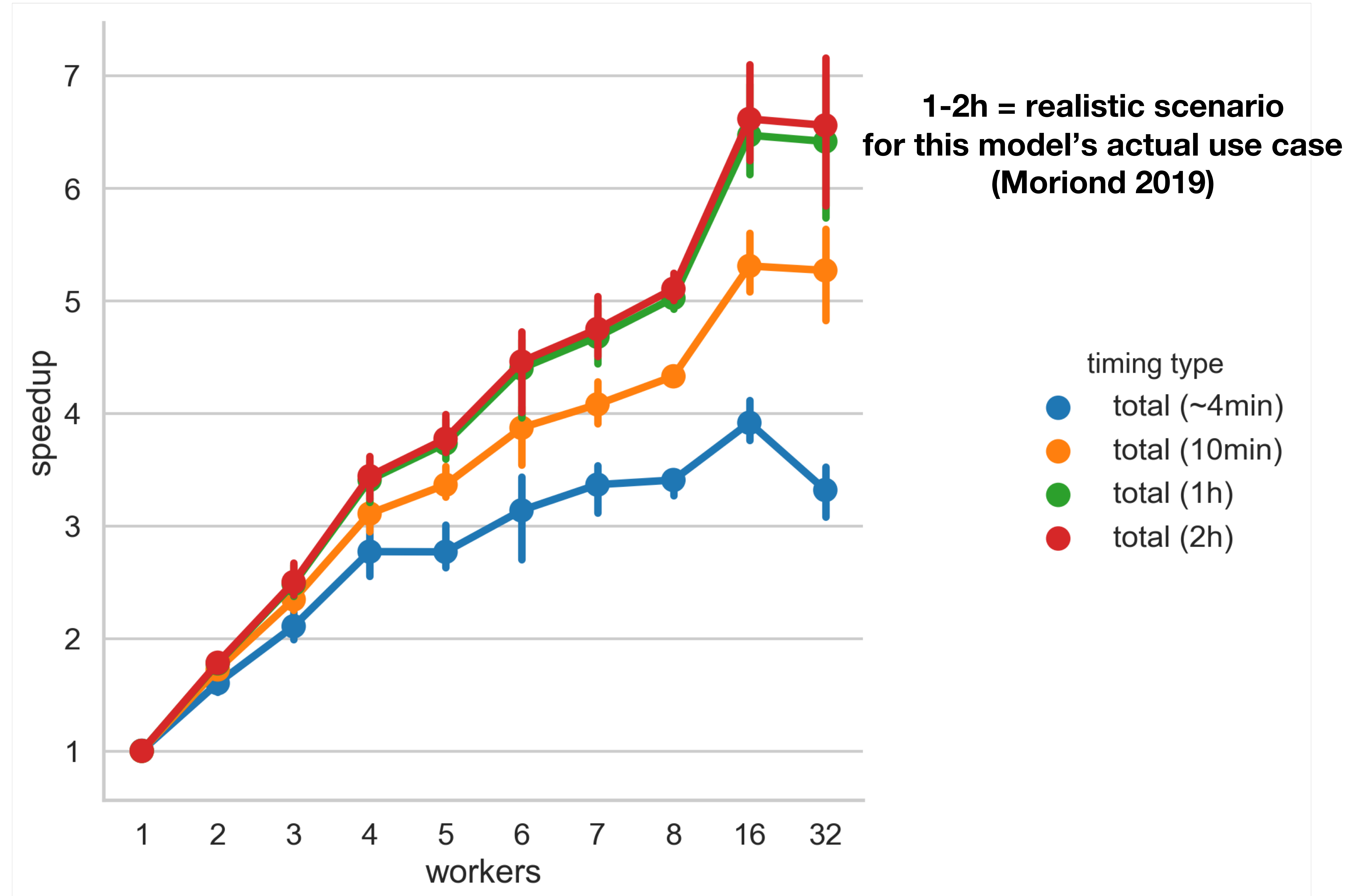
1 2 3 4 5 6 7 8 16 32

workers

Extrapolate big model scaling results to longer runs

Speedup for fits starting further from minimum

- Extrapolations, not actual measurements
- Assume overhead fixed



Conclusions

Interactive study of complex LHC physics fits (e.g. Higgs) requires short fitting wall times (minutes rather than hours) → Parallelization can deliver this

We improved scaling performance of existing likelihood-level parallelization

Introduced new flexible framework: multi-level parallelization (likelihood, gradient)

Gradient-level parallelization scales for large (> 1h) fits = main goal

Future work

Bring new infrastructure into production version of ROOT/RooFit
for the adventurous: development version @ github.com/roofit-dev/root

Investigate imperfect scaling observed in gradient calculation.

Redesign core RooFit test statistic classes to make future-proof interface
allows to plug in any new types of calculation strategy (e.g. anal. derivatives)
HESSE can be parallelized in similar way
incorporate orthogonal work by Stephan on vectorization

With combination of all techniques expect speedups of factor 20 for fits >1 hour.

Let’s stay in touch

+31 (0)6 10 79 58 74

p.bos@esciencecenter.nl

www.esciencecenter.nl

egpbos

linkedin.com/in/egpbos

blog.esciencecenter.nl

netherlands eScience center

Encore

Load balancing

PDF timings change dynamically due to RooFit precalculation strategies

... not a problem for numerical integrals

Analytical derivatives (automated? `CLAD`)

Minuit confidence intervals

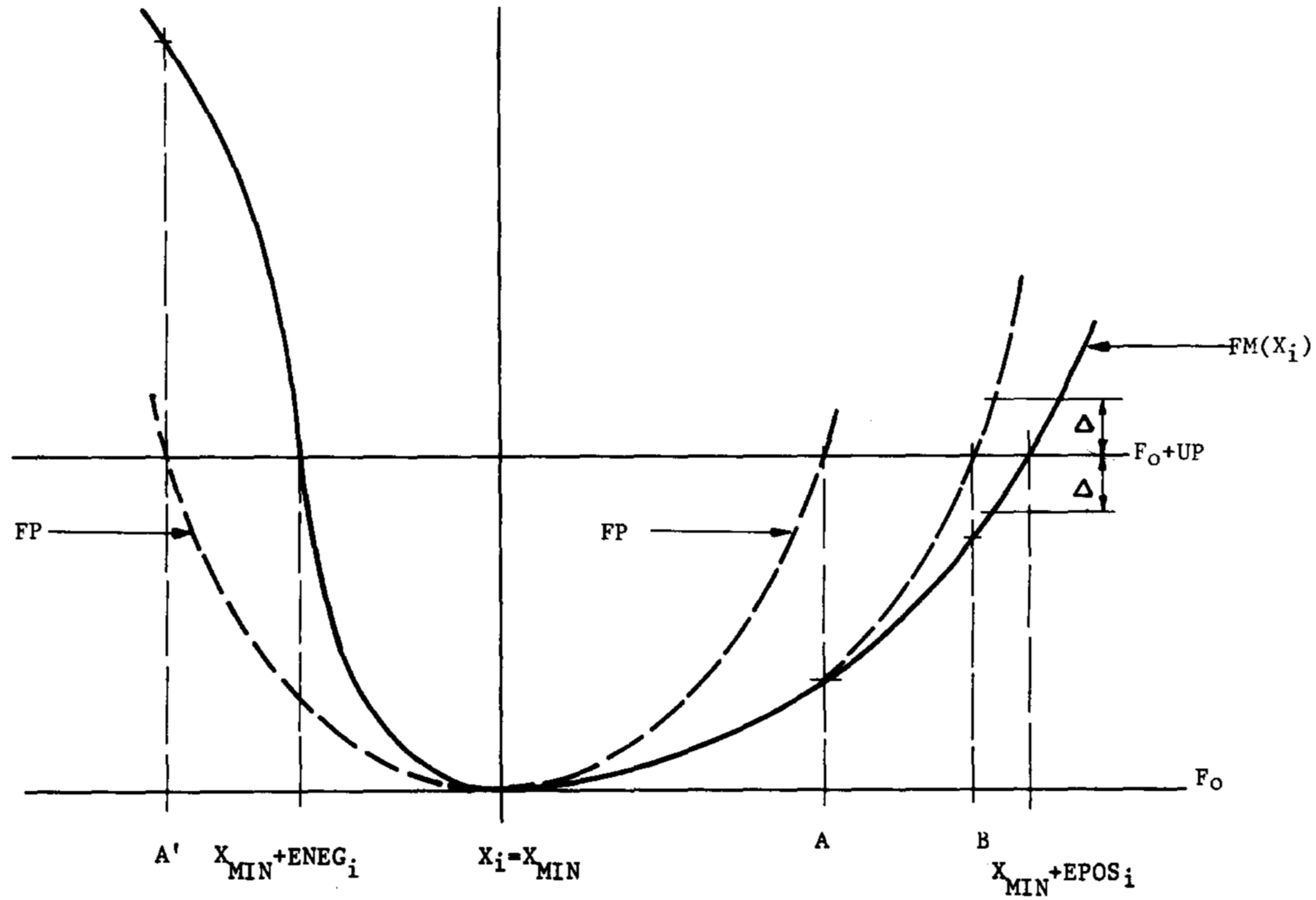
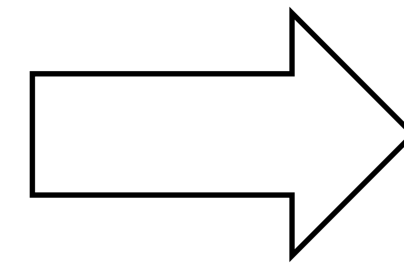


Fig. 2. Calculation of MINOS errors of parameter *i*. The (symmetric) dotted parabola FP is predicted from the covariance matrix, but the nonlinearity of the problem results in the solid curve FM which gives the asymmetric errors EPOS and ENEG (see text).

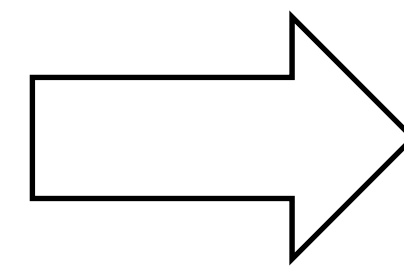
Numerical integrals

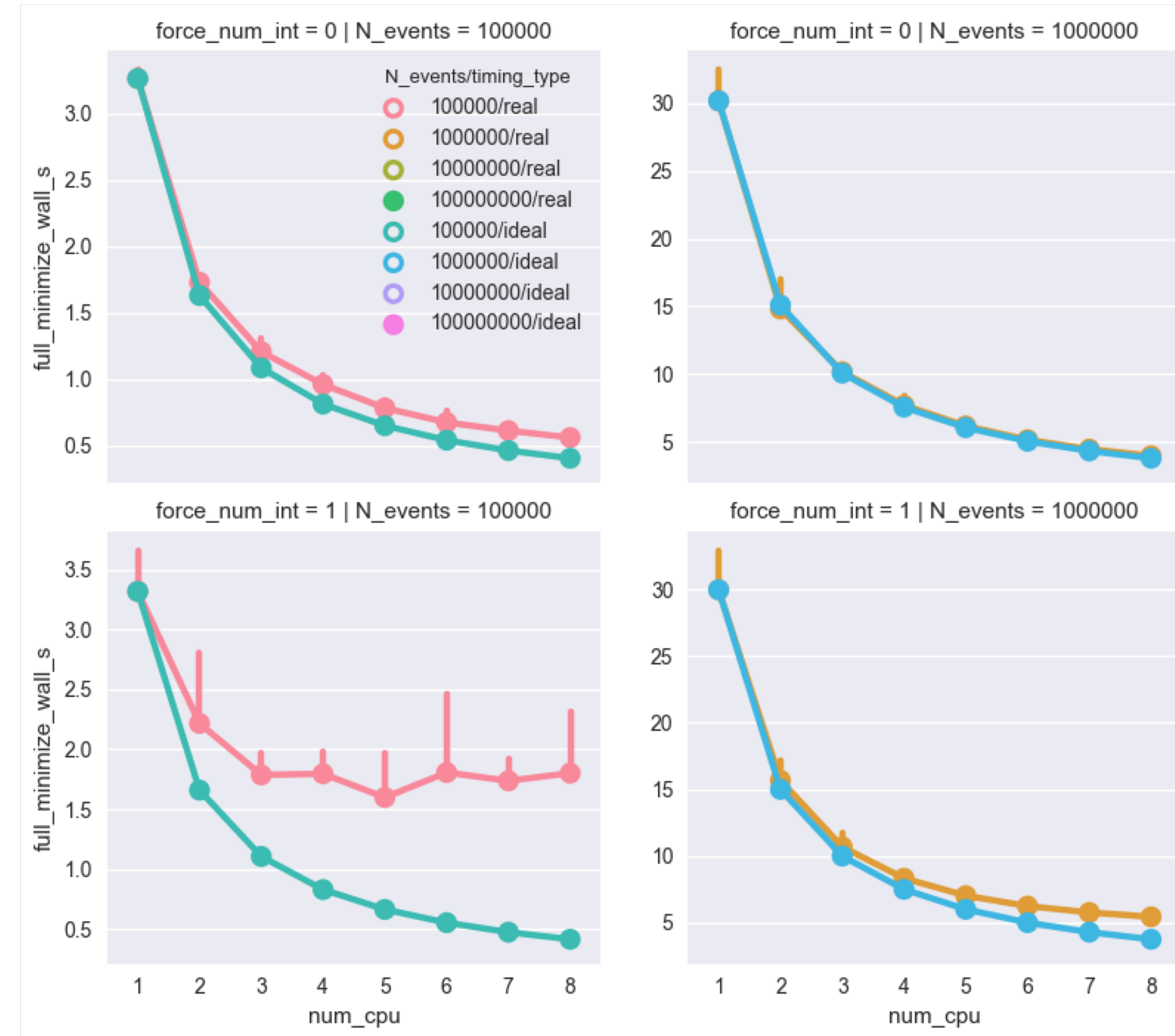
“Analytical” integrals



Forced numerical (Monte Carlo) integrals

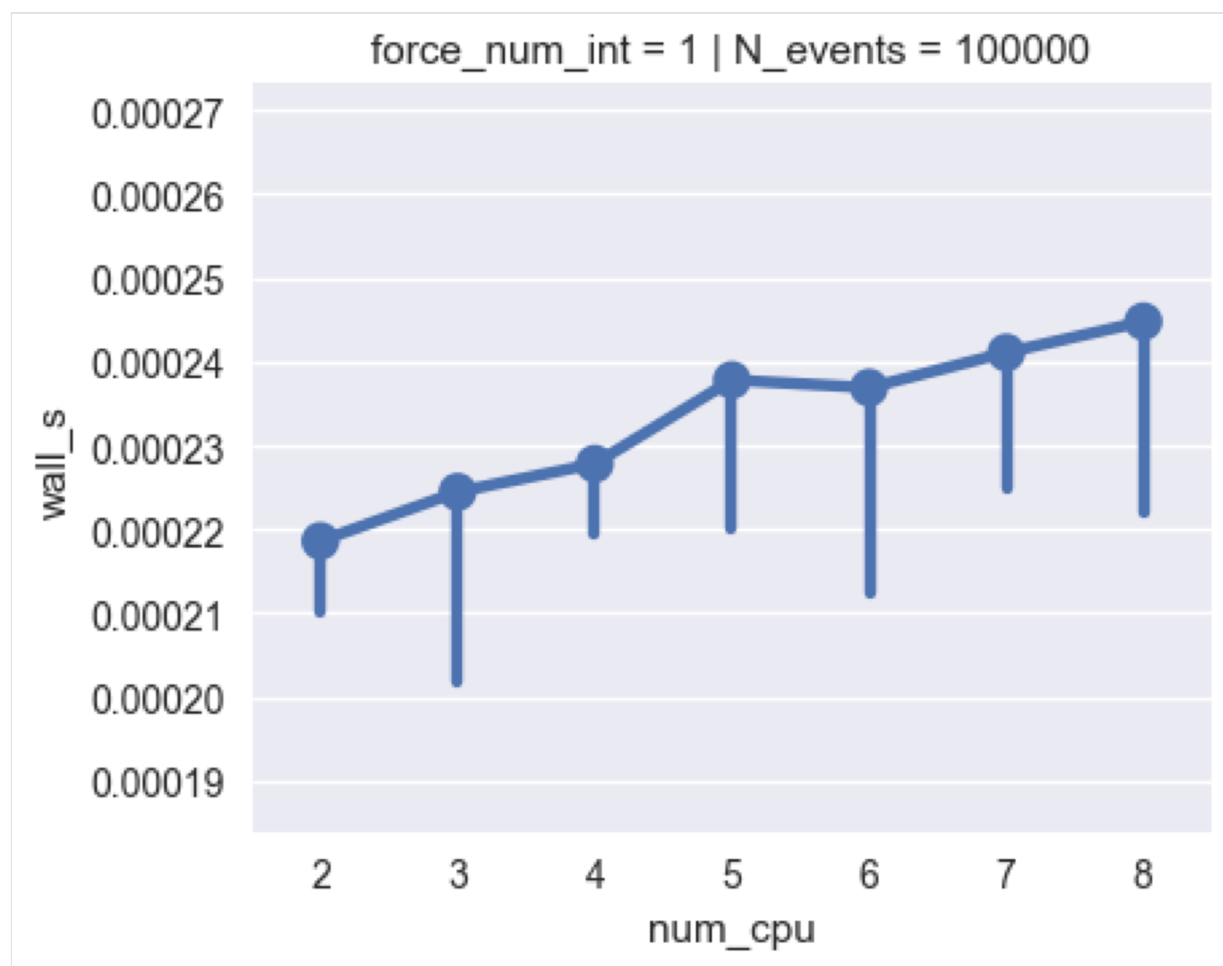
(Higgs fits didn’t have them)



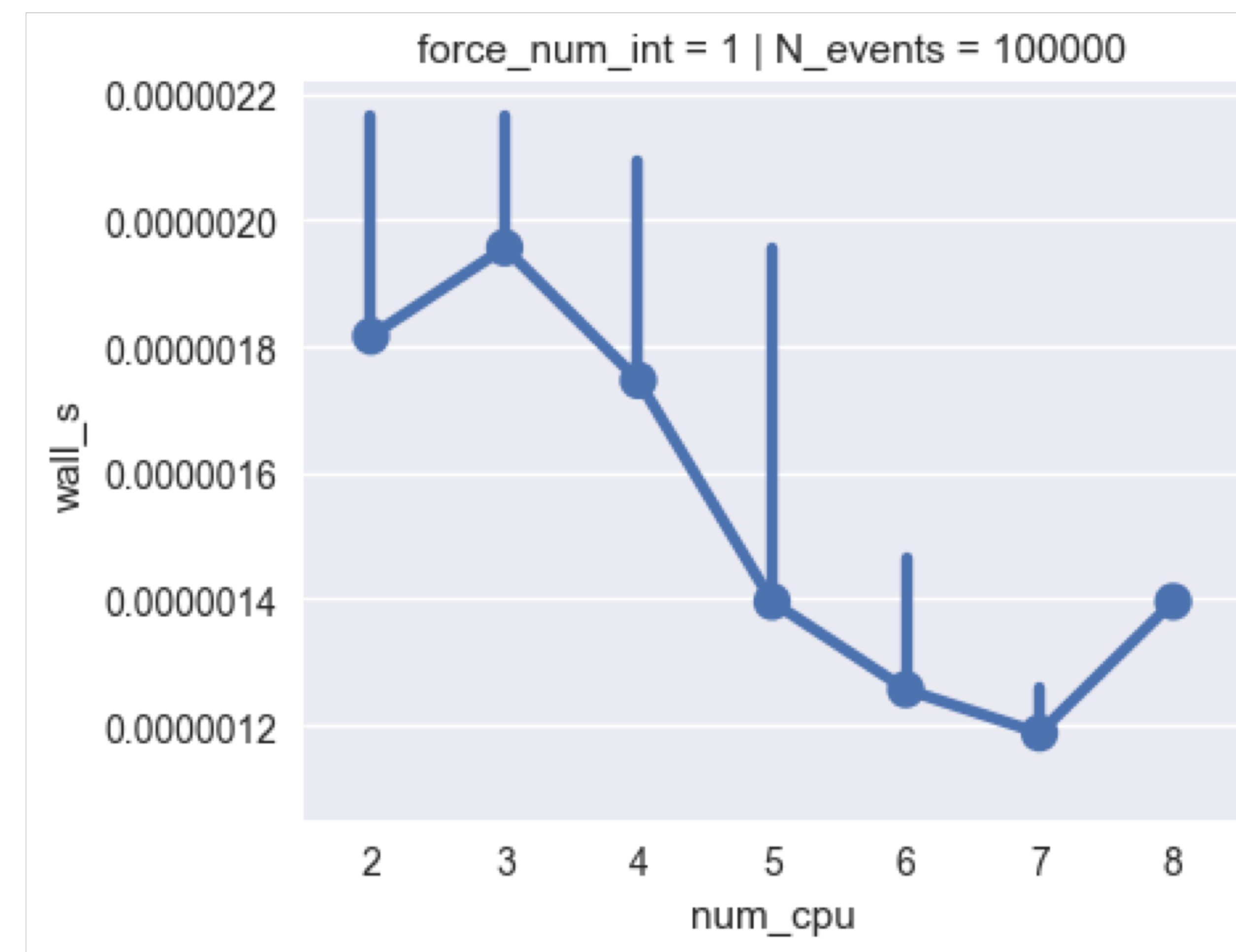


Individual NI timings
(variation in runs and iterations)

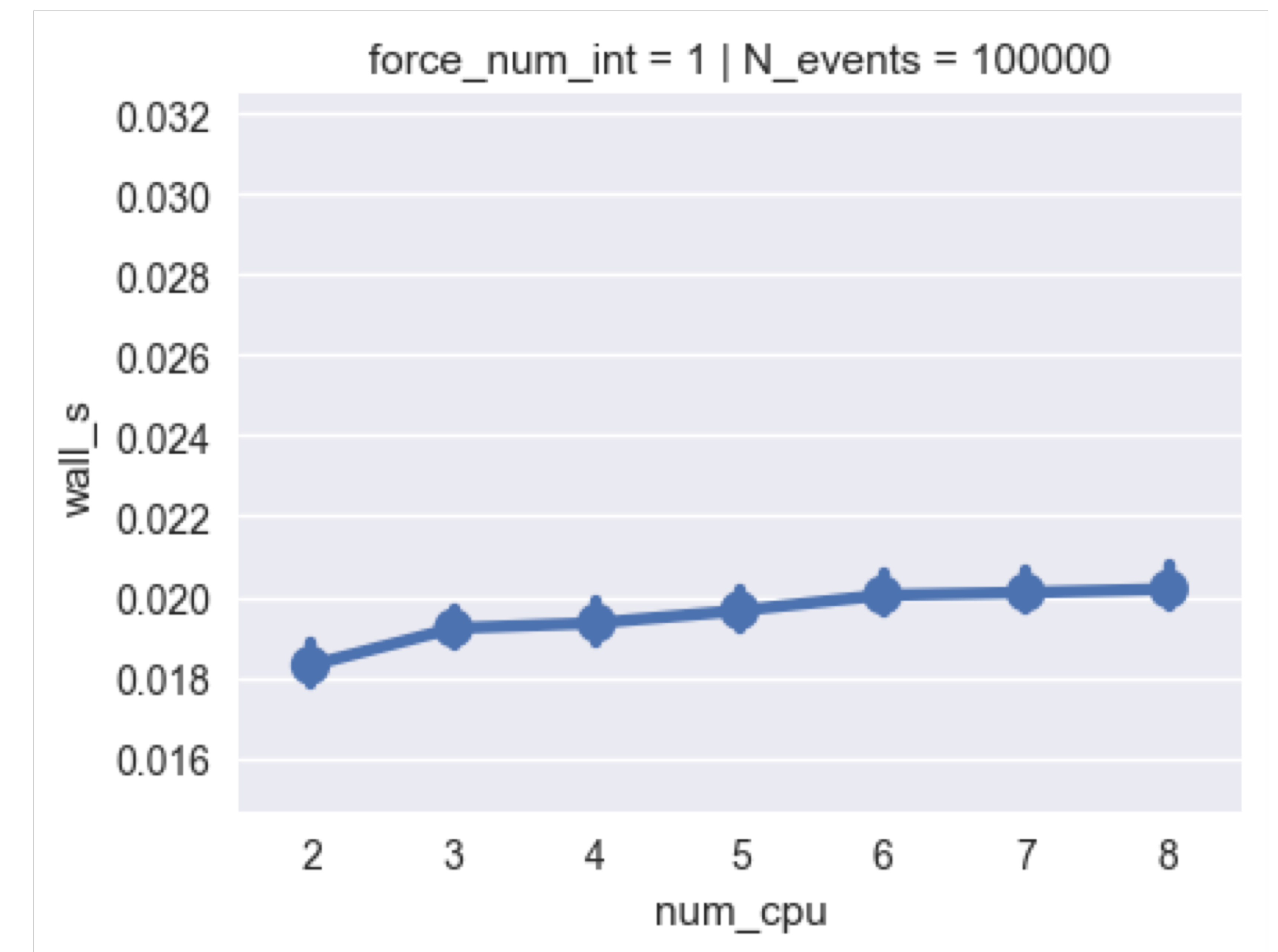
Maxima



Minima



Sum of slowest integrals/cores
per iteration over the entire run



(single core total runtime: 3.2s)

`RooFit::MultiProcess::Vector<YourSerialClass>`

Serial class: likelihood (e.g. `RooNLLVar`) or gradient (Minuit)

Interface: subclass + MP

Define ”vector elements”

Group elements into tasks (to be executed in parallel)

`RooFit::MultiProcess::SharedArg<T>`

`RooFit::MultiProcess::TaskManager`

`RooFit::MultiProcess::Vector<YourSerialClass>`

`RooFit::MultiProcess::SharedArg<T>`

Normalization integrals or other shared expensive objects

Parallel task definition specific to type of object

… design in progress

`RooFit::MultiProcess::TaskManager`

Faster fitting: MultiProcess design

RooFit::MultiProcess::Vector<YourSerialClass>

RooFit::MultiProcess::SharedArg<T>

RooFit::MultiProcess::TaskManager

Queue gathers tasks and communicates with worker pool

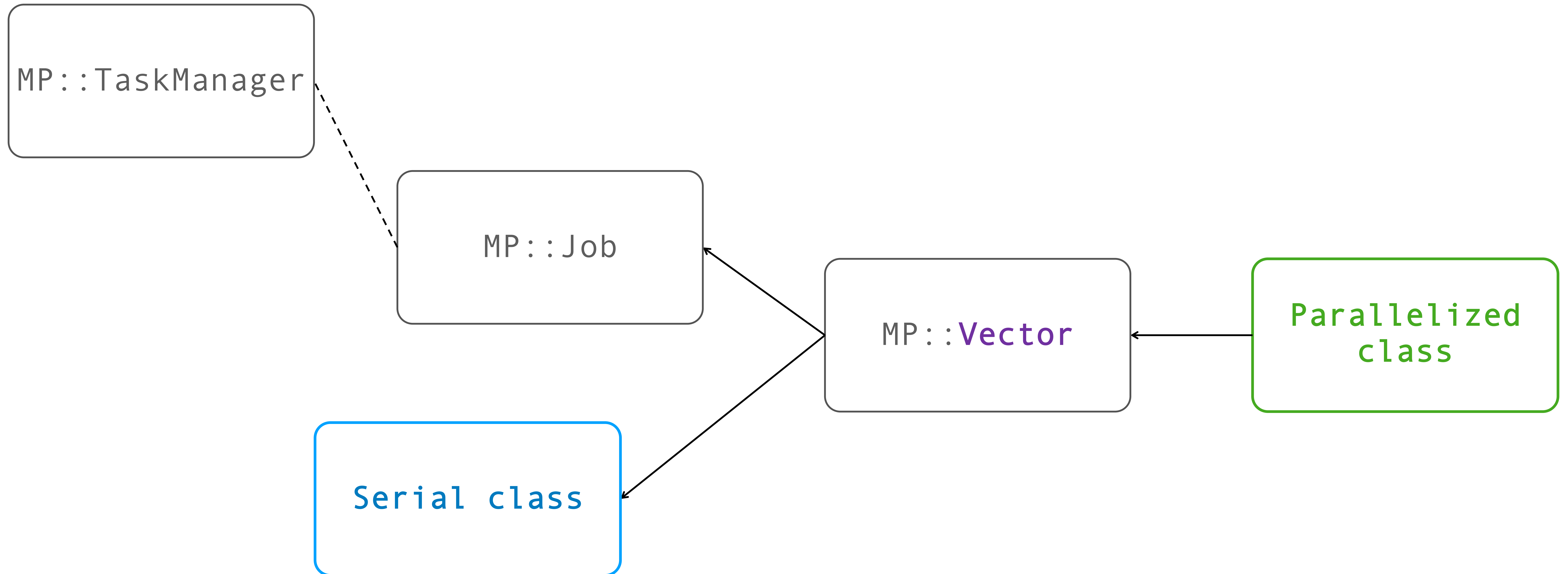
Workers steal tasks from queue

Worker pool: forked processes (BidirMMapPipe)

- performant and already used in RooFit
- no thread-safety concerns
- instead: communication concerns
- … flexible design, implementation can be replaced (e.g. TBB)

```
vector<double> x {1, 4, 5, 6.48074};

xSquaredSerial xsq_serial(x);

size_t N_workers = 4;
xSquaredParallel xsq_parallel(N_workers, x);

// get the same results, but now faster:
xsq_serial.get_result();
xsq_parallel.get_result();

// use parallelized version in your existing functions
void some_function(xSquaredSerial* xsq);

some_function(&xsq_parallel);  // no problem!
```

MultiProcess usage for devs




```
template <class T> class MP::Vector : public T, public MP::Job
class Parallel : public MP:Vector<Serial>
```

MultiProcess usage for devs

```
class xSquaredSerial {
 public:
  xSquaredSerial(vector<double> x_init)
          : x(move(x_init))
          , result(x.size()) {}

  virtual void evaluate() {
    for (size_t ix = 0; ix < x.size(); ++ix) {
      x_squared[ix] = x[ix] * x[ix];
    }
  }

  vector<double> get_result() {
    evaluate();
    return x_squared;
  }

 protected:
  vector<double> x;
  vector<double> x_squared;
};
```

```
class xSquaredParallel
  : public RooFit::MultiProcess::Vector<xSquaredSerial> {
 public:
  xSquaredParallel(size_t N_workers, vector<double> x_init) :
      RooFit::MultiProcess::Vector<xSquaredSerial>(N_workers, x_init)
  {}

 private:
  void evaluate_task(size_t task) override {
    result[task] = x[task] * x[task];
  }

 public:
  void evaluate() override {
    if (get_manager()->is_master()) {
      // do necessary synchronization before work_mode

      // enable work mode: workers will start stealing work from queue
      get_manager()->set_work_mode(true);

      // master fills queue with tasks
      for (size_t task_id = 0; task_id < x.size(); ++task_id) {
        get_manager()->to_queue(JobTask(id, task_id));
      }

      // wait for task results back from workers to master
      gather_worker_results();

      // end work mode
      get_manager()->set_work_mode(false);

      // put gathered results in desired container (same as used in serial class)
      for (size_t task_id = 0; task_id < x.size(); ++task_id) {
        x_squared[task_id] = results[task_id];
      }
    }
  }
};
```

```
template <class T> class MP::Vector : public T, public MP::Job
```

Single core profiling and improvements

Higgs ggf & 9 channel fits (workspaces by Lydia Brenner)

Most time spent on:

1. Memory access → RooVectorDataStore::get() (4% / 32%), 0.3% LL cache misses (expensive!)
   - Row-wise access pattern on column-wise data store (and std::vector<std::vector>)
2. Logarithms: 12%
3. Interpolation → RooStats::HistFactory::FlexibleInterpVar (10%)

Faster fitting: single core improvements

RooLinkedList::findArg: ~ 5% of memory access instructions

RooLinkedList::At took considerable time in Gaussian test fit (*Vince*)

std::vector lookup → 1.6x speedup! WIP

Reorder tree evaluation → CPU cache use, vectorization

Smarter fitting (stochastic minimizer, analytical gradient, CLAD)

Front-end / back-end separation (e.g. TensorFlow back-end)

Faster fitting: single core profiling meta-conclusions

- profiling functions & classes
  - valgrind
  - gprof
  - Instruments
  - … etc.
- profiling objects (e.g. call-trees, e.g. RooFit…)
  - … DIY?

More Multi-Core

`RooRealMPFE` / `BidirMMapPipe`

Custom multi-process message passing protocol

- POSIX `fork`, `pipe`, `mmap`

Communication “overhead” (delay between sending and receiving messages): ~ 1e-4 seconds

- `serverLoop` waits for message & runs server-side code
- messages used sparingly
- data transfer over memory-mapped pipes

	RooFit (MINUIT)	TensorFlow (BFGS)
Unbinned fit	0.1s	**0.01 - 0.1s** (dep. on precision)
Binned fit	**0.7ms**	2.3ms

Fits on identical model & data (single i7 machine)

TensorFlow: No pre-calculation / caching!

- Major advantage of RooFit for binned fits (e.g. morphing histograms)
- (feature request for memoization https://github.com/tensorflow/tensorflow/issues/5323)

N.B.: measured before CPU affinity fixing

- RooFit now even faster (but limited to running one machine)